DOCUMENT RESUME

ED 232 860 SE 042 634

AUTHOR	Novak, Gordon S., Jr.
TITLE	GLISP User's Manual. Revised.
INSTITUTION	Stanford Univ., Calif. Dept. of Computer Science.
SPONS AGENCY	Advanced Research Projects Agency (DOD), Washington, D.C.; National Science Foundation, Washington, D.C.
PUB DATE	23 Nov 82
CONTRACT	MDA-903-80-c-007
GRANT	SED-7912803
NOTE	43p.; For related documents, see SE 042 630-635.
PUB TYPE	Guides - General (050) -- Reference Materials - General (130)
EDRS PRICE	MF01/PC02 Plus Postage.
DESCRIPTORS	*Computer Programs; *Computer Science; Guides; *Programing; *Programing Languages; *Resource Materials
IDENTIFIERS	*GLISP Programing Language; National Science Foundation

ABSTRACT

GLISP is a LISP-based language which provides high-level language features not found in ordinary LISP. The GLISP language is implemented by means of a compiler which accepts GLISP as input and produces ordinary LISP as output. This output can be further compiled to machine code by the LISP compiler. GLISP is available for several LISP dialects, including Interlisp, Maclisp, UCI Lisp, ELISP, Franz Lisp, and Portable Standard Lisp. The goal of GLISP is to allow structured objects to be referenced in a convenient, succinct language and to allow the structures of objects to be changed without changing the code which references the objects. GLISP provides both PASCAL-like and English-like syntaxes; much of the power and brevity of GLISP derive from the compiler features necessary to support the relatively informal, English-like language constructs. Provided in this manual is the documentation necessary for using GLISP. The documentation is presented in the following sections: introduction; object descriptions; reference to objects; GLISP program syntax; messages; context rules and reference; GLISP and knowledge representation languages; obtaining and using GLISP; GLISP hacks (some ways of doing things in GLISP which might not be entirely obvious at first glance); and examples of GLISP object declarations and programs. (JN)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

U.S. DEPARTMENT OF EDUCATION
NATIONAL INSTITUTE OF EDUCATION
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)
✓ This document has been reproduced as received from the person or organization originating it.
☐ Minor changes have been made to improve reproduction quality.
• Points of view or opinions stated in this document do not necessarily represent official NIE position or policy.

ED232860

GLISP User's Manual

Gordon S. Novak Jr.
Computer Science Department
Stanford University
Stanford, California 94305

Revised: 23 November 1982

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY
National Science Foundation
TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

This research was supported in part by NSF grant SED-7912803 in the Joint National Science Foundation - National Institute of Education Program of Research on Cognitive Processes and the Structure of Knowledge in Science and Mathematics, and in part by the Defense Advanced Research Projects Agency under contract MDA-903-80-c-007.

SE042634

Table of Contents

1. Introduction 1

1.1. Overview of GLISP 1
1.2. Implementation 2
1.3. Error Messages 3
1.4. Interactive Features of GLISP 3

2. Object Descriptions 5

2.1. Declaration of Object Descriptions 5
2.1.1. Property Descriptions 6
2.1.2. Supers Description 6
2.1.3. Values Description 6
2.2. Structure Descriptions 7
2.2.1. Syntax of Structure Descriptions 7
2.2.2. Examples of Structure Descriptions 9
2.3. Editing of Object Descriptions 9
2.4. Interactive Editing of Objects 10
2.5. Global Variables 10
2.6. Compile-Time Constants and Conditional Compilation 10

3. Reference To Objects 13

3.1. Accessing Objects 13
3.2. Creation of Objects 14
3.3. Interpretive Creation of Objects 14
3.4. Predicates on Objects 14
3.4.1. Self-Recognition Adjectives 15
3.4.2. Testing Object Classes 15

4. GLISP Program Syntax 17

4.1. Function Syntax 17
4.2. Expressions 18
4.2.1. Interpretation of Operators 19
4.2.1.1. Operations on Strings 19
4.2.1.2. Operations on Lists 19
4.2.1.3. Compound Operators 19
4.2.1.4. Assignment 20
4.2.1.5. Self-Assignment Operators[39]
4.3. Control Statements 21
4.3.1. IF Statement 22
4.3.2. CASE Statement 22
4.3.3. FOR Statement 22
4.3.4. WHILE Statement 23
4.3.5. REPEAT Statement 23

[39]This section may be skipped by the casual user of GLISP. 21

4.4. Definite Reference to Particular Objects 24

5. Messages 25

5.1. Compilation of Messages 25
5.2. Compilation of Properties and Adjectives 27
5.3. Declarations for Message Compilation 27
5.4. Operator Overloading 28
5.5. Runtime Interpretation of Messages 28

6. Context Rules and Reference 31

6.1. Organization of Context 31
6.2. Rules for Using Definite Reference 31
6.3. Type Inference 32

7. GLISP and Knowledge Representation Languages 33

8. Obtaining and Using GLISP 35

8.1. Documentation 35
8.2. Compiler Files 35
8.3. Getting Started 35
8.4. Reserved Words and Characters 36
8.4.1. Reserved Characters 36
8.4.2. Reserved Function Names 36
8.4.3. Other Reserved Names 37
8.5. Lisp Dialect Idiosyncrasies 37
8.5.1. Interlisp 37
8.5.2. UCI Lisp 37
8.5.3. ELISP 37
8.5.4. Maclisp 37
8.5.5. Franz Lisp 37
8.6. Bug Reports and Mailing List 38

9. GLISP Hacks 39

9.1. Overloading Basic Types 39
9.2. Disjunctive Types 40
9.3. Generators 40

10. Program Examples 43

10.1. GLTST1 File 43
10.2. GLTST2 File 43

1. Introduction

1.1. Overview of GLISP

GLISP is a LISP-based language which provides high-level language features not found in ordinary LISP. The GLISP language is implemented by means of a compiler which accepts GLISP as input and produces ordinary LISP as output; this output can be further compiled to machine code by the LISP compiler. GLISP is available for several LISP dialects, including Interlisp, Maclisp, UCI Lisp, ELISP, Franz Lisp, and Portable Standard Lisp.

The goal of GLISP is to allow structured objects to be referenced in a convenient, succinct language, and to allow the structures of objects to be changed without changing the code which references the objects. GLISP provides both PASCAL-like and English-like syntaxes; much of the power and brevity of GLISP derive from the compiler features necessary to support the relatively informal, English-like language constructs. The following example function illustrates how GLISP permits definite reference to structured objects.

```
(HourlySalaries (GLAMBDA ( (a DEPARTMENT) )
   (for each EMPLOYEE who is HOURLY
      (PRIN1 NAME) (SPACES 3) (PRINT SALARY) )  ))
```

The features provided by GLISP include the following:

1. GLISP maintains knowledge of the "context" of the computation as the program is executed. Features of objects which are in context may be referenced directly; the compiler will determine how to reference the objects given the current context, and will add the newly referenced objects to the context. In the above example, the function's argument, an object whose class is DEPARTMENT, establishes an initial context relative to which EMPLOYEEs can be found. In the context of an EMPLOYEE, NAME and SALARY can be found.

2. GLISP supports flexible object definition and reference with a powerful abstract datatype facility. Object classes are easily declared to the system. An object declaration includes a definition of the storage structure of the object and declarations of properties of the object; these may be declared in such a way that they compile open, resulting in efficient object code. GLISP supports object-centered programming, in which processes are invoked by means of "messages" sent to objects. Object structures may be LISP structures (for which code is automatically compiled) or Units in the user's favorite representation language (for which the user can supply compilation functions).

3. Loop constructs, such as (FOR EACH <item> WITH <property> DO ...), are compiled into loops of the appropriate form.

4. Compilation of infix expressions is provided for the arithmetic operators and for additional operators which facilitate list manipulation. Operators are interpreted appropriately for Lisp datatypes as well as for numbers; operator overloading for user-defined objects is provided using the message facility.

5. The GLISP compiler infers the types of objects when possible, and uses this knowledge to generate efficient object code. By performing *compilation relative to a knowledge base*, GLISP is able to perform certain computations (e.g., inheritance of an attached procedure from a parent class of an object in a knowledge base) at compile time rather than at runtime, resulting in much faster execution.

6. By separating object definitions from the code which references objects, GLISP permits radical changes to object structures with no changes to code.

1.2. Implementation

GLISP is implemented by means of a compiler, which produces a normal Lisp EXPR from the GLISP code; the GLISP code is saved on the function's property list, and the compiled definition replaces the GLISP definition. Use of GLISP entails the cost of a single compilation, but otherwise is about as efficient as normal LISP. The LISP code produced by GLISP can be further compiled to machine code by the LISP compiler.

GLISP functions are indicated by the use of GLAMBDA instead of LAMBDA in the function definition. When the Lisp interpreter sees the GLAMBDA, it calls the GLISP compiler to incrementally compile the GLISP function. The compiled version replaces the GLISP version (which is saved on the function name's property list), and is used thereafter. This automatic compilation feature is currently implemented in Interlisp and in Franz Lisp. In other dialects, it is necessary for the user to explicitly invoke compilation of GLISP functions by calling the compiler function `GLCC` for each one.

To use GLISP, it is first necessary to load the compiler file into Lisp. Users' files containing structure descriptions and GLISP code are then loaded. Compilation of a GLISP function is requested by:

`(GLCC 'FN)`	Compile `FN`.
`(GLCP 'FN)`	Compile `FN` and prettyprint the result.
`(GLP 'FN)`	Print the compiled version of `FN`.

In Interlisp, all the GLISP functions (beginning with GLAMBDA) in a file can be compiled by invoking `(GLCOMPCOMS <file>COMS)`, where `<file>COMS` is the list of file package commands for the file.

Properties of compiled functions are stored on the property list of the function name:

`GLORIGINALEXPR`	Original (GLISP) version of the function.[1]
`GLCOMPILED`	GLISP-compiled version of the function.

[1] The original definition is saved as EXPR in Interlisp.

GLRESULTTYPE	Type of the result of the function.
GLARGUMENTTYPES	Types of the arguments of the function.

Properties of GLISP functions can be examined with the function (GLED '<name>), which calls the Lisp editor on the property list of <name>. (GLEDF '<name>) calls the Lisp editor on the original (GLISP) definition of <name>.

1.3. Error Messages

GLISP provides detailed error messages when compilation errors are detected; many careless errors such as misspellings will be caught by the compiler. When the source program contains errors, the compiled code generates runtime errors upon execution of the erroneous expressions.

1.4. Interactive Features of GLISP

Several features of GLISP are available interactively, as well as in compiled functions:

1. The A function, which creates structured objects from a readable property/value list, is available as an interactive function.

2. Messages to objects can be executed interactively.

3. A display editor/inspector, GEV, is available for use with bitmap graphics terminals.[2] GEV interprets objects according to their GLISP structure descriptions; it allows the user to inspect objects, edit them, interactively construct programs which operate on them, display computed properties, send messages to objects, and "push down" to inspect data values.

[2]GEV is currently implemented only for Xerox Lisp machines.

2. Object Descriptions

2.1. Declaration of Object Descriptions

An *Object Description* in GLISP is a description of the structure of an object in terms of named substructures, together with definitions of ways of referencing the object. The latter may include *properties* (i.e., data whose values are not stored, but are computed from the values of stored data), adjectival predicates, and *messages* which the object can receive; the messages can be used to implement operator overloading and other compilation features.

Object Descriptions are obtained by GLISP in several ways:

1. The descriptions of basic datatypes (e.g., INTEGER) are automatically known to the compiler.
2. Structure descriptions (but not full object descriptions) may be used directly as *types* in function definitions.
3. The user may declare object descriptions to the system using the function GLISPOBJECTS; the names of the object types may then be used as *types* in function definitions and definitions of other structures.
4. Object descriptions may be included as part of a knowledge representation language, and are then furnished to GLISP by the interface package written for that representation language.

LISP data structures are declared using the function GLISPOBJECTS[3], which takes one or more object descriptions as arguments (assuming the descriptions to be quoted). Since GLISP compilation is performed relative to the knowledge base of object descriptions, the object descriptions must be declared prior to GLISP compilation of functions using those descriptions. The format of each description is as follows:

```
(<object name>   <structure description>
          PROP   <property descriptions>
          ADJ    <adjective descriptions>
          ISA    <predicate descriptions>
          MSG    <message descriptions>
          SUPERS <list of superclasses>
          VALUES <list of values>              )
```

The <object name> and <structure description> are required; the other property/value pairs are optional, and may appear in any order. The following example illustrates some of the declarations which might be made to describe the object type VECTOR.

---

[3]Once declared, object descriptions may be included in INTERLISP program files by including in the <file>COMS a statement of the form: (GLISPOBJECTS <object-name$_1$> ... <object-name$_n$>)

```
(GLISPOBJECTS

   (VECTOR   (CONS (X NUMBER) (Y NUMBER))

      PROP   ( (MAGNITUDE  ((SQRT X*X + Y*Y))) )

      ADJ    ( (ZERO       (X IS ZERO AND Y IS ZERO))
               (NORMALIZED (MAGNITUDE = 1.0)) )

      MSG    ( (+          VECTORPLUS OPEN T)
               (-          VECTORDIFFERENCE) )

     ))
```

2.1.1. Property Descriptions

Each <description> specified with PROP, ADJ, ISA, or MSG has the following format:

(<name> <response> $<prop_1>$ $<value_1>$ ... $<prop_n>$ $<value_n>$)

where <name> is the (atomic) name of the property, <response> is a function name or a list of GLISP code to be compiled in place of the property, and the <prop> <value> pairs are optional properties which affect compilation. All four kinds of properties are compiled in a similar fashion, as described in the section "Compilation of Messages".

2.1.2. Supers Description

The SUPERS list specifies a list of *superclasses*, i.e., the names of other object descriptions from which the object may inherit PROP, ADJ, ISA, and MSG properties. Inheritance from superclasses can be recursive, as described under "Compilation of Messages".

2.1.3. Values Description

The VALUES list is a list of pairs, (<name> <value>), which is used to associate symbolic names with constant values for an object type. If VALUES are defined for the type of the *selector* of a CASE statement, the corresponding symbolic names may be used as the selection values for the clauses of the CASE statement.

2.2. Structure Descriptions

Much of the power of GLISP is derived from its use of Structure Descriptions. A Structure Description (abbreviated "<sd>") is a means of describing a LISP data structure and giving names to parts of the structure; it is similar in concept to a Record declaration in PASCAL. Structure descriptions are used by the GLISP compiler to generate code to retrieve and store parts of structures.

2.2.1. Syntax of Structure Descriptions

The syntax of structure descriptions is recursively defined in terms of basic types and composite types which are built up from basic types. The syntax of structure descriptions is as follows:[4]

1. The following basic types are known to the compiler:

ATOM	
INTEGER	
REAL	
NUMBER	(either INTEGER or REAL)
STRING	
BOOLEAN	(either T or NIL)
ANYTHING	(an arbitrary structure)

2. An object type which is known to the compiler, either from a GLISPOBJECTS declaration or because it is a Class of units in the user's knowledge representation language, is a valid type for use in a structure description. The <name> of such an object type may be specified directly as <name> or, for readability, as **(A** <name>**)** or **(AN** <name>**)**.[5]

3. Any substructure can be named by enclosing it in a list prefixed by the name: **(**<name> <sd>**)**. This allows the same substructure to have multiple names. "A", "AN", and the names used in forming composite types (given below) are treated as reserved words, and may not be used as names.

4. Composite Structures: Structured data types composed of other structures are described using the following structuring operators:

   a. **(CONS** <sd_1> <sd_2>**)**
   The CONS of two structures whose descriptions are <sd_1> and <sd_2>.

   b. **(LIST** <sd_1> <sd_2> ... <sd_n>**)**
   A list of exactly the elements whose descriptions are <sd_1> <sd_2> ... <sd_n>.

   c. **(LISTOF** <sd>**)**

---

[4]The names of the basic types and the structuring operators must be all upper-case or lower-case, depending on the case which is usual for the underlying Lisp system. In general, other GLISP keywords and user program names may be in upper-case, lower-case, or mixed-case, if mixed cases are permitted by the Lisp system.

[5]Whenever the form (A ...) is allowed in GLISP, the form (AN ...) is also allowed.

A list of zero or more elements, each of which has the description <sd>.

d. (ALIST (<name$_1$> <sd$_1$>) ... (<name$_n$> <sd$_n$>))
An association list in which the atom <name$_i$>, if present, is associated with a structure whose description is <sd$_i$>.

e. (PROPLIST (<name$_1$> <sd$_1$>) ... (<name$_n$> <sd$_n$>))
An association list in "property-list format" (alternating names and values) in which the atom <name$_i$>, if present, is associated with a structure whose description is <sd$_i$>.

f. (ATOM (BINDING <sd>) (PROPLIST (<pname$_1$> <sd$_1$>) ... (<pname$_n$> <sd$_n$>)))
This describes an atom with its binding and/or its property list; either the BINDING or the PROPLIST group may be omitted. Each property name <pname$_i$> is treated as a property list indicator as well as the name of the substructure. When creation of such a structure is specified, GLISP will compile code to create a GENSYM atom.

g. (RECORD <recordname> (<name$_1$> <sd$_1$>) ... (<name$_n$> <sd$_n$>))
RECORD specifies the use of contiguous records for data storage. <recordname> is the name of the record type; it is optional, and is not used in some Lisp dialects.[6]

h. (TRANSPARENT <type>)
An object of type <type> is incorporated into the structure being defined in *transparent mode*, which means that all fields and properties of the object of type <type> can be directly referenced as if they were properties of the object being defined. A substructure which is a named *type* and which is not declared to be TRANSPARENT is assumed to be opaque, i.e., its internal structure cannot be seen unless an access path explicitly names the subrecord.[7] The object of type <type> may also contain TRANSPARENT objects; the graph of TRANSPARENT object references must of course be acyclic.

i. (OBJECT (<name$_1$> <sd$_1$>) ... (<name$_n$> <sd$_n$>))
(ATOMOBJECT (<name$_1$> <sd$_1$>) ... (<name$_n$> <sd$_n$>))
(LISTOBJECT (<name$_1$> <sd$_1$>) ... (<name$_n$> <sd$_n$>))
These declarations describe *Objects*, data structures which can receive messages at runtime. The three types of objects are implemented as records, atoms, or lists, respectively. In each case, the system adds to the object a CLASS datum which points to the name of the type of the object. An object declaration may only appear as the top-level declaration of a named object type.

---

[6]RECORDs are implemented using RECORDs in Interlisp, HUNKs in Maclisp and Franz Lisp, VECTORs in Portable Standard Lisp, and lists in UCI Lisp and ELISP. In Interlisp, appropriate RECORD declarations must be made to the system by the user in addition to the GLISP declarations.

[7]For example, a PROFESSOR record might contain some fields which are unique to professors, plus a pointer to an EMPLOYEE record. If the declaration in the PROFESSOR record were `(EMPREC (TRANSPARENT EMPLOYEE))`, then a field of the employee record, say SALARY, could be referenced directly from a variable P which points to a PROFESSOR record as `P:SALARY` ; if the declaration were `(EMPREC EMPLOYEE)`, it would be necessary to say `P:EMPREC:SALARY`.

2.2.2. Examples of Structure Descriptions

The following examples illustrate the use of Structure Descriptions.

```
(GLISPOBJECTS

    (CAT (LIST (NAME ATOM)
               (PROPERTIES (LIST (CONS (SEX ATOM)
                                       (WEIGHT INTEGER))
                                 (AGE INTEGER)
                                 (COLOR ATOM)))
               (LIKESCATNIP BOOLEAN)))

    (PERSON (ATOM
             (PROPLIST.
               (CHILDREN (LISTOF (A PERSON)))
               (AGE INTEGER)
               (PETS (LIST (CATS (LISTOF CAT))
                           (DOGS (LISTOF (A DOG))) ))
            )))
)
```

The first structure, CAT, is entirely composed of list structure. An CAT structure might look like:

```
(PUFF ((MALE . 10) 5 CALICO) T)
```

Given a CAT object X, we could ask for its WEIGHT [equivalent to (CDAADR X)] or for a subrecord such as PROPERTIES [equivalent to (CADR X)]. Having set a variable Y to the PROPERTIES, we could also ask for the WEIGHT from Y [equivalent to (CDAR Y)]. In general, whenever a subrecord is accessed, the structure description of the subrecord is associated with it by the compiler, enabling further accesses to parts of the subrecord. Thus, the meaning of a subrecord name depends on the type of record from which the subrecord is retrieved. The subrecord AGE has two different meanings when applied to PERSONs and CATs. The second structure, PERSON, illustrates a description of an object which is a Lisp atom with properties stored on its property list. Whereas no structure names appear in an actual CAT structure, the substructures of a PROPLIST operator must be named, and the names appear in the actual structures. For example, if X is a PERSON structure, retrieval of the AGE of X is equivalent to `(GETPROP X 'AGE)`. A subrecord of a PROPLIST record can be referenced directly; e.g., one can ask for the DOGS of a PERSON directly, without cognizance of the fact that DOGS is part of the PETS property.

2.3. Editing of Object Descriptions

An object description can be edited by calling `(GLEDS TYPE)`, where `TYPE` is the name of the object type. This will cause the Lisp editor to be called on the object description of `TYPE`.

2.4. Interactive Editing of Objects

An interactive structure inspector/editor, GEV, is available for the Xerox 1100-series lisp machines. GEV allows the user to inspect and edit any structures which are described by GLISP object descriptions, to "zoom in" on substructures of interest, and to display the values of computed properties automatically or on demand. GEV is described in a separate document.

2.5. Global Variables

The types of free variables can be declared within the functions which reference them. Alternatively, the types of global variables can be declared to the compiler using the form:[8]

```
(GLISPGLOBALS  (<name> <type>) ... )
```

Following such a declaration, the compiler will assume a free variable <name> is of the corresponding <type>. A GLOBAL object does not have to actually exist as a storage structure; for example, one could define a global object "MOUSE" or "SYSTEM" whose properties are actually implemented by calls to the operating system.

2.6. Compile-Time Constants and Conditional Compilation

The values and types of compile-time constants can be declared to the compiler using the form:[9]

```
(GLISPCONSTANTS  (<name> <value-expression> <type>) ... )
```

The <name> and <type> fields are assumed to be quoted. The `<value-expression>` field is a GLISP expression which is parsed and evaluated; this allows constants to be defined by expressions involving previously defined constants.

The GLISP compiler will perform many kinds of computations on constants at compile time, reducing the size of the compiled code and improving execution speed.[10] In particular, arithmetic, comparison, logical, conditional, and CASE function calls are optimized, with elimination of dead code. This permits conditional compilation in a clean form. Code can be written which tests the values of flags in the usual way; if the flag values are then declared to be compile-time constants using GLISPCONSTANTS, the tests will be performed

---

[8] (GLISPGLOBALS $\langle name_1 \rangle$ ... $\langle name_n \rangle$) is defined as a file package command for Interlisp.

[9] (GLISPCONSTANTS $\langle name_1 \rangle$ ... $\langle name_n \rangle$) is defined as a file package command for Interlisp.

[10] Ordinary Lisp functions are evaluated on constant arguments if the property GLEVALWHENCONST is set to T on the property list of the function name. This property is set by the compiler for the basic arithmetic functions.

at compile time, and the unneeded code will vanish.

3. Reference To Objects

3.1. Accessing Objects

The problem of reference is the problem of determining what object, or feature of a structured object, is referred to by some part of a statement in a language. Most programming languages solve the problem of reference by unique naming: each distinct object in a program unit has a unique name, and is referenced by that name. Reference to a part of a structured object is done by giving the name of the variable denoting that object and a path specification which tells how to get to the desired part from the whole.

GLISP permits reference by unique naming and path specification, but in addition permits *definite reference relative to context.* A *definite reference* is a reference to an object which has not been explicitly named before, but which can be understood relative to the current context of computation. If, for example, an object of type VECTOR (as defined earlier) is in context, the program statement

```
(IF X IS NEGATIVE ...
```

contains a definite reference to "X", which may be interpreted as the X substructure of the VECTOR which is in context. The definition of the computational context and the way in which definite references are resolved are covered in a later section of this manual.

In the following section, which describes the syntaxes of reference to objects in GLISP, the following notation is used. "<var>" refers to a variable name in the usual LISP sense, i.e., a LAMBDA variable, PROG variable, or GLOBAL variable; the variable is assumed to point to (be bound to) an object. "<type>" refers to the type of object pointed to by a variable. "<property>" refers to a property or subrecord of an object.

Two syntaxes are available for reference to objects: an English-like syntax, and a PASCAL-like syntax. The two are equivalent, and may be intermixed freely within a GLISP function. The allowable forms of references in the two syntaxes are shown in the table below.

"PASCAL" Syntax	"English" Syntax	Meaning
<var>	<var>	The object denoted by <var>
:<type>	The <type>	The object whose type is <type>
:<property> *or* <property>	The <property>	The <property> of some object
<var>:<property>	The <property> of <var>	The <property> of the object denoted by <var>

These forms can be extended to specify longer paths in the obvious way, as in "The AGE of the SPOUSE of

the HEAD of the DEPARTMENT" or "DEPARTMENT:HEAD:SPOUSE:AGE". Note that there is no distinction between reference to substructures and reference to properties as far as the syntax of the referencing code is concerned; this facilitates hiding the internal structures of objects.

3.2. Creation of Objects

GLISP allows the creation of structures to be specified by expressions of the form:

(A <type> *with* <property$_1$> = <value$_1$>, ... , <property$_n$> = <value$_n$>)

In this expression, the "*with*", "=", and "," are allowed for readability, but may be omitted if desired[11]; if present, they must all be delimited on both sides by blanks. In response to such an expression, GLISP will generate code to create a new instance of the specified structure. The <property> names may be specified in any order. Unspecified properties are defaulted according to the following rules:

1. Basic types are defaulted to 0 for INTEGER and NUMBER, 0.0 for REAL, and NIL for other types.

2. Composite structures are created from the defaults of their components, except that missing PROPLIST and ALIST items which would default to NIL are omitted.

Except for missing PROPLIST and ALIST elements, as noted above, a newly created LISP structure will contain all of the fields specified in its structure description.

3.3. Interpretive Creation of Objects

The "A" function is defined for interpretive use as well as for use within GLISP functions.

3.4. Predicates on Objects

Adjectives defined for structures using the `ADJ` and `ISA` specifications may be used in predicate expressions on objects in **If** and **For** statements. The syntax of basic predicate expressions is:

```
<object> is <adjective>
<object> isa <isa-adjective>
```

Basic predicate expressions may be combined using AND, OR, NOT or ~, and grouping parentheses.

The compiler automatically recognizes the LISP adjectives ATOMIC, NULL, NIL, INTEGER, REAL, ZERO, NUMERIC, NEGATIVE, MINUS, and BOUND, and the ISA-adjectives ATOM, LIST, NUMBER,

---

[11] Some Lisp dialects, e.g. Maclisp, will interpret commas as "backquote" commands and generate error messages. In such dialects, the commas must be omitted or be "slashified".

INTEGER, SYMBOL, STRING, ARRAY, and BIGNUM[12]; user definitions have precedence over these pre-defined adjectives.

3.4.1. Self-Recognition Adjectives

If the ISA-adjective `self` is defined for an object type, the type name may be used as an ISA-adjective to test whether a given object is a member of that type. Given a predicate phrase of the form " `X is a Y` ", the compiler first looks at the definition of the object type of `X` to see if `Y` is defined as an ISA-adjective for such objects. If no such ISA-adjective is found, and `Y` is a type name, the compiler looks to see if `self` is defined as an ISA-adjective for `Y` , and if so, compiles it.

If a `self` ISA-adjective predicate is compiled as the test of an **If**, **While**, or **For** statement, and the tested object is a simple variable, the variable will be known to be of that type within the scope of the test. For example, in the statement

```
(If X is a FOO then (← X Print) ...
```

the compiler will know that X is a FOO if the test succeeds, and will compile the Print message appropriate for a FOO, even if the type of X was declared as something other than FOO earlier. This feature is useful in implementing disjunctive types, as discussed in a later section.

3.4.2. Testing Object Classes

For those data types which are defined using one of the OBJECT structuring operators, the Class name is automatically defined as an ISA-adjective. The ISA test is implemented by runtime examination of the CLASS datum of the object.

---

[12] where applicable.

4. GLISP Program Syntax

4.1. Function Syntax

GLISP function syntax is essentially the same as that of LISP with the addition of type information and RESULT and GLOBAL declarations. The basic function syntax is:[13]

```
(<function-name> (GLAMBDA (<arguments>)
                          (RESULT <result-description>)
                          (GLOBAL <global-variable-descriptions>)
      (PROG (<prog-variables>)
            <code>   )))
```

The RESULT declaration is optional; in many cases, the compiler will infer the result type automatically. The main use of the RESULT declaration is to allow the compiler to determine the result type without compiling the function, which may be useful when compiling another function which calls it. The <result-description> is a standard structure description or <type>.

The GLOBAL declaration is used to inform the compiler of the types of free variables. The function GLISPGLOBALS can be used to declare the types of global variables, making GLOBAL declarations within individual functions unnecessary.

The major difference between a GLISP function definition and a standard LISP definition is the presence of type declarations for variables, which are in PASCAL-like syntax of the following forms:

```
<variable>:<type>
<variable>:(A <type>)
<variable>,<variable>,...:<type>
<variable>,<variable>,...:(A <type>)
          :<type>
           (A <type>)
```

In addition to declared <type>s, a Structure Description may be used directly as a <type> in a variable declaration.

Type declarations are required only for variables whose subrecords or properties will be referenced. In general, if the value of a variable is computed in such a way that the type of the value can be inferred, the variable will receive the appropriate type automatically; in such cases, no type declaration is necessary. Since GLISP maintains a *context* of the computation, it is often unnecessary to name a variable which is an

---

[13] The PROG is not required. In Lisp dialects other than Interlisp, LAMBDA may be used instead of GLAMBDA.

argument of a function; in such cases, it is only necessary to specify the <type> of the argument, as shown in the latter two syntax forms above. PROG and GLOBAL declarations must always specify variable names (with optional types); the ability to directly reference features of objects reduces the number of PROG variables needed in many cases.

Initial values for PROG variables may be specified, as in Interlisp, by enclosing the variable and its initial value in a list[14]:

```
(PROG (X (N 0) Y) ...)
```

However, the syntax of variable declarations does not permit the type of a variable and its initial value to both be specified.

4.2. Expressions

GLISP provides translation of infix expressions of the kind usually found in programming languages. In addition, it provides additional operators which facilitate list manipulation and other operations. Overloading of operators for user-defined types is provided by means of the *message* facility.

Expressions may be written directly in-line within function references, as in `(SQRT X*X + Y*Y)` , or they may be written within parentheses; parentheses may be used for grouping in the usual way. Operators may be written with or without delimiting spaces, *except for the "-" operator, which* ***must*** *be delimited by spaces.*[15] Expression parsing is done by an operator precedence parser, using the same precedence ordering as in FORTRAN.[16] The operators which are recognized are as follows:[17]

Assignment	← *or* :=
Arithmetic	+ - * / ↑
Comparison	= ~= <> < <= > >=
Logical	AND OR NOT ~
Compound	←+ ←- +← -←

---

[14] This feature is available in all Lisp dialects.

[15] The "-" operator is required to be delimited by spaces since "-" is often used as a hyphen within variable names. The "-" operator will be recognized within "atom" names if the flag GLSEPMINUS is set to T.

[16] The precedence of compound operators is higher than assignment but lower than that of all other operators. The operators ↑ ← ←+ +← ←- -← are right-associative; all others are left-associative.

[17] In Maclisp, the operator / must be written //.

4.2.1. Interpretation of Operators

In addition to the usual interpretation of operators when used with numeric arguments, some of the operators are interpreted appropriately for other Lisp types.

4.2.1.1. Operations on Strings

For operands of type STRING, the operator + performs concatenation. All of the comparison operators are defined for STRINGs.

4.2.1.2. Operations on Lists

Several operators are defined in such a way that they perform set operations on lists of the form (LISTOF <type>) , where <type> is considered to be the element type. The following table shows the interpretations of the operators:

<list> + <list>	Set Union
<list> - <list>	Set Difference
<list> * <list>	Set Intersection

<list>	+	<element>	CONS
<element>	+	<list>	CONS
<list>	-	<element>	REMOVE
<element>	<=	<list>	MEMBER or MEMB
<list>	>=	<element>	MEMBER or MEMB

4.2.1.3. Compound Operators

Each compound operator performs an operation involving the arguments of the operator and assigns a value to the left-hand argument; compound operators are therefore thought of as "destructive change" operators. The meaning of a compound operator depends on the type of its left-hand argument, as shown in the following table:

Operator	Mnemonic	NUMBER	LISTOF	BOOLEAN
←+	*Accumulate*	PLUS	NCONC1	OR
←-	*Remove*	DIFFERENCE	REMOVE	AND NOT
+←	*Push*	PLUS	PUSH	OR
-←	*Pop*		POP[18]	

As an aid in remembering the list operators, the arrow may be thought of as representing the list, with the head of the arrow being the front of the list and the operation (+ or -) appearing where the operation occurs on the list. Thus, for example, ←+ adds an element at the end of the list, while +← adds an element at the front of the list.

---

[18] For the Pop operator, the arguments are in the reverse of the usual order, i.e., (TOP -← STACK) will pop the top element off STACK and assign the element removed to TOP.

Each of the compound operators performs an assignment to its left-hand side; the above table shows an abbreviation of the operation which is performed prior to the assignment. The following examples show the effects of the operator "←+" on local variables of different types:

Type	Source Code	Compiled Code
INTEGER	`I ←+ 5`	`(SETQ I (IPLUS I 5))`
BOOLEAN	`P ←+ Q`	`(SETQ P (OR P Q))`
LISTOF	`L ←+ ITEM`	`(SETQ L (NCONC1 L ITEM))`

When the compound operators are not specifically defined for a type, they are interpreted as specifying the operation (+ or -) on the two operands, followed by assignment of the result to the left-hand operand.

4.2.1.4. Assignment

Assignment of a value to the left-hand argument of an assignment operator is relatively flexible in GLISP. The following kinds of operands are allowed on the left-hand side of an assignment operator:

1. Variables.
2. Stored substructures of a structured type.
3. PROPerties of a structured type, whenever the interpretation of the PROPerty would be a legal left-hand side.
4. Algebraic expressions involving numeric types, *provided* that the expression ultimately involves only one occurrence of a variable or stored value.[19]

For example, consider the following Object Description for a CIRCLE:

```
(CIRCLE (LIST (START VECTOR) (RADIUS REAL))
  PROP  ((PI            (3.1415926))
         (DIAMETER      (RADIUS*2))
         (CIRCUMFERENCE (PI*DIAMETER))
         (AREA          (PI*RADIUS↑2))) )
```

Given this description, and a CIRCLE `C`, the following are legal assignments:

```
(C:RADIUS ← 5.0)
(C:AREA ← 100.0)
(C:AREA ← C:AREA*2)
(C:AREA ←+ 100.0)
```

---

[19] For example, `(X↑2 ← 2.0)` is acceptable, but `(X*X ← 2.0)` is not because the variable `X` occurs twice.

4.2.1.5. Self-Assignment Operators[20]

There are some cases where it would be desirable to let an object perform an assignment of its own value. For example, the user might want to define *PropertyList* as an abstract datatype, with messages such as GETPROP and PUTPROP, and use PropertyLists as substructures of other datatypes. However, a message such as PUTPROP may cause the PropertyList object to modify its own structure, perhaps even changing its structure from NIL to a non-NIL value. If the function which implements PUTPROP performs a normal assignment to its "self" variable, the assignment will affect only the local variable, and will not modify the PropertyList component of the containing structure. The purpose of the Self-Assignment Operators is to allow such modification of the value within the containing structure.

The Self-Assignment Operators are ←←, ←←+, ←←+, and ←←-, corresponding to the operators ←, ←+, +←, and ←-, respectively. The meaning of these operators is that the assignment is performed to the object on the left-hand side of the operator, *as seen from the structure containing the object.*

The use of these operators is highly restricted; any use of a Self-Assignment Operator must meet all of the following conditions:

1. A Self-Assignment Operator can only be used within a Message function which is compiled OPEN.

2. The left-hand side of the assignment must be a simple variable which is an argument of the function.

3. The left-hand-side variable must be given a unique (unusual) name to prevent accidental aliasing with a user variable name.

As an example, the PUTPROP message for a PropertyList datatype could be implemented as follows:

```
(PropertyList.PUTPROP (GLAMBDA (PropertyListPUTPROPself prop val)
    (PropertyListPUTPROPself ←←
            (LISTPUT PropertyListPUTPROPself prop val)) ))
```

4.3. Control Statements

GLISP provides several PASCAL-like control statements.

---

[20] This section may be skipped by the casual user of GLISP.

4.3.1. IF Statement

The syntax of the IF statement is as follows:

```
(IF       <condition1> THEN <action11> ... <action1i>
ELSEIF    <condition2> THEN <action21> ... <action2j>
...
ELSE      <actionm1> ... <actionmk>)
```

Such a statement is translated to a COND of the obvious form. The "THEN" keyword is optional, as are the "ELSEIF" and "ELSE" clauses.

4.3.2. CASE Statement

The CASE statement selects a set of actions based on an atomic selector value; its syntax is:

```
(CASE     <selector> OF
          (<case1> <action11> ... <action1i>)
          (<case2> <action21> ... <action2j>)
          ...
          ELSE    <actionm1> ... <actionmk>)
```

The <selector> is evaluated, and is compared with the given <case> specifications. Each <case> specification is either a single, atomic specification, or a list of atomic specifications. All <case> specifications are assumed to be quoted. The "ELSE" clause is optional; the "ELSE" actions are executed if <selector> does not match any <case>.

If the *type* of the <selector> has a VALUES specification, <case> specifications which match the VALUES for that type will be translated into the corresponding values.

4.3.3. FOR Statement

The FOR statement generates a loop through a set of elements (typically a list). Two syntaxes of the FOR statement are provided:

```
(FOR EACH <set> DO <action1> ... <actionn>)

(FOR <variable> IN <set> DO <action1> ... <actionn>)
```

The keyword "DO" is optional. In the first form of the FOR statement, the singular form of the <set> is specified; GLISP will convert the given set name to the plural form.[21] The <set> may be qualified by an adjective or predicate phrase in the first form; the allowable syntaxes for such qualifying phrases are shown below:

---

[21] For names with irregular plurals, the plural form should be put on the property list of the singular form under the property name PLURAL, e.g., (PUTPROP 'MAN 'PLURAL 'MEN).

```
<set> WITH <predicate>
<set> WHICH IS <adjective>
<set> WHO IS   <adjective>
<set> THAT IS  <adjective>
```

The <predicate> and <adjective> phrases may be combined with AND, OR, NOT, and grouping parentheses. These phrases may be followed by a qualifying phrase of the form:

```
WHEN <expression>
```

The "WHEN" expression is ANDed with the other qualifying expressions to determine when the loop body will be executed.

Within the FOR loop, the current member of the <set> which is being examined is automatically put into *context* at the highest level of priority. For example, suppose that the current context contains a substructure whose description is:

```
(PLUMBERS (LISTOF EMPLOYEE))
```

Assuming that EMPLOYEE contains the appropriate definitions, the following FOR loop could be written:

```
(FOR EACH PLUMBER WHO IS NOT A TRAINEE DO SALARY ← + 1.50)
```

To simplify the collection of features of a group of objects, the <action>s in the FOR loop may be replaced by the CLISP-like construct:

```
... COLLECT <form>)
```

4.3.4. WHILE Statement

The format of the WHILE statement is as follows:

```
(WHILE <condition> DO <action1> ... <actionn>)
```

The actions <action$_1$> through <action$_n$> are executed repeatedly as long as <condition> is true. The keyword DO may be omitted. The value of the expression is NIL.

4.3.5. REPEAT Statement

The format of the REPEAT statement is as follows:

```
(REPEAT <action1> ... <actionn> UNTIL <condition>)
```

The actions <action$_1$> through <action$_n$> are repeated (always at least once) until <condition> is true. The value of the expression is NIL. The keyword UNTIL is required.

4.4. Definite Reference to Particular Objects

In order to simplify reference to particular member(s) of a group, definite reference may be used. Such an expression is written using the word THE followed by the singular form of the group, or THOSE followed by the plural form of the group, and qualifying phrases (as described for the FOR statement). The following examples illustrate these expressions.

```
(THE SLOT WITH SLOTNAME = NAME)
(THOSE EMPLOYEES WITH JOBTITLE = 'ELECTRICIAN)
```

The value of THE is a single object (or NIL if no object satisfies the specified conditions); THOSE produces a list of all objects satisfying the conditions.[22]

[22] In general, nested loops are optimized so that intermediate lists are not actually constructed. Therefore, use of nested THE or THOSE statements is not inefficient.

5. Messages

GLISP supports the *Message* metaphor, which has its roots in the languages SIMULA and SMALLTALK. These languages provide *Object-Centered Programming*, in which objects are thought of as being active entities which communicate by sending each other *Messages*. The internal structures of objects are hidden; a program which wishes to access "variables" of an object does so by sending messages to the object requesting the access desired. Each object contains[23] a list of *Selectors*, which identify the messages to which the object can respond. A *Message* specifies the destination object, the selector, and any arguments associated with the message. When a message is executed at runtime, the selector is looked up for the destination object; associated with the selector is a procedure, which is executed with the destination object and message arguments as its arguments.

GLISP treats reference to properties, adjectives, and predicates associated with an object similarly to the way it treats messages. The compiler is able to perform much of the lookup of *selectors* at compile time, resulting in efficient code while maintaining the flexibility of the message metaphor. Messages can be defined in such a way that they compile open, compile as function calls to the function which is associated with the selector, or compile as messages to be interpreted at runtime.

Sending of a *message* in GLISP is specified using the following syntax:

(SEND <object> <selector> <arg$_1$> ... <arg$_n$>)

The keyword "SEND" may be replaced by "←". The <selector> is assumed to be quoted. Zero or more arguments may be specified; the arguments other than <selector> are evaluated. <object> is evaluated; if <object> is a non-atomic expression, it must be enclosed in at least one set of parantheses, so that the <selector> will always be the third element of the list.

5.1. Compilation of Messages

When GLISP encounters a message statement, it looks up the <selector> in the MSG definition of the type of the object to which the message is sent, or in one of the SUPERS of the type.[24] Each <selector> is paired with the appropriate <response> to the message. Code is compiled depending on the form of the <response> associated with the <selector>, as follows:[25]

---

[23] typically by inheritance from some parent in a Class hierarchy

[24] If an appropriate representation language is provided, the <selector> and its associated <response> may be inherited from a parent class in the class hierarchy of the representation language.

[25] If the type of the destination object is unknown, or if the <selector> cannot be found, GLISP compiles the (SEND ...) statement as if it is a normal function call.

1. If the <response> is an atom, that atom is taken as the name of a function which is to be called in response to the message. The code which is compiled is a direct call to this function,

   (<response> <object> <arg$_1$> ... <arg$_n$>)

2. If the <response> is a list, the contents of the list are recursively compiled in-line as GLISP code, with the name "self" artificially "bound" to the <object> to which the message was sent. Because the compilation is recursive, a message may be defined in terms of other messages, substructures, or properties, which may themselves be defined as messages.[26] The outer pair of parentheses of the <response> serves only to bound its contents; thus, if the <response> is a function call, the function call must be enclosed in an additional set of parentheses.

The following examples illustrate the various ways of defining message responses.

```
(EDIT          EDITV)

(SUCCESSOR     (self + 1))

(MAGNITUDE     ((SQRT X*X + Y*Y)))
```

In the first example, a message with <selector> EDIT is compiled as a direct call to the function EDITV. In the second example, the SUCCESSOR message is compiled as the sum of the object receiving the message (represented by "self") and the constant 1; if the object receiving the message is the value of the variable J and has the type INTEGER, the code generated for the SUCCESSOR would be (ADD1 J). The third example illustrates a call to a function, SQRT, with arguments containing definite references to X and Y (which presumably are defined as part of the object whose MAGNITUDE is sought). Note that since MAGNITUDE is defined by a function call, an "extra" pair of parentheses is required around the function call to distinguish it from in-line code.

The user can determine whether a message is to be compiled open, compiled as a function call, or compiled as a message which is to be executed at runtime. When a GLISP expression is specified as a <response>, the <response> is always compiled open; open compilation can be requested by using the OPEN property when the <response> is a function name. Open compilation operates like macro expansion; since the "macro" is a GLISP expression, it is easy to define messages and properties in terms of other messages and properties. The combined capabilities of open compilation, message inheritance, conditional compilation, and flexible assignment provide a great deal of power. The ability to use definite reference in GLISP makes the definition and use of the "macros" simple and natural.

---

[26] Such recursive definitions must of course be acyclic.

5.2. Compilation of Properties and Adjectives

Properties, Adjectives, and ISA-adjectives are compiled in the same way as Messages. Since the syntax of use of properties and adjectives does not permit specification of any arguments, the only argument available to code or a function which implements the <response> for a property or adjective is the `self` argument, which denotes the object to which the property or adjective applies. A <response> which is written directly as GLISP code may use the name `self` directly[27], as in the SUCCESSOR example above; a function which is specified as the <response> will be called with the `self` object as its single argument.

5.3. Declarations for Message Compilation

Declarations which affect compilation of Messages, Adjectives, or Properties may be specified following the <response> for a given message; such declarations are in (Interlisp) property-list format, <prop$_1$><value$_1$> ... <prop$_n$><value$_n$>. The following declarations may be specified:

1. RESULT <type>
   This declaration specifies the *type* of the result of the message or other property. Specification of result types helps the compiler to perform type inference, thus reducing the number of type declarations needed in user programs. The RESULT type for simple GLISP expressions will be inferred by the compiler; the RESULT declaration should be used if the <response> is a complex GLISP expression or a function name.[28]

2. OPEN T
   This declaration specifies that the function which is specified as the <response> is to be compiled open at each reference. A <response> which is a list of GLISP code is always compiled open; however, such a <response> can have only the `self` argument. If it is desired to compile open a Message <response> which has arguments besides `self`, the <response> must be coded as a function (in order to bind the arguments) and the OPEN declaration must be used. Functions which are compiled open may not be recursive via any chain of open-compiled functions.

3. MESSAGE T
   This declaration specifies that a runtime message should be generated for messages with this <selector> sent to objects of this Class. Typically, such a declaration would be used in a higher-level Class whose subclasses have different responses to the same message <selector>.

---

[27] The name `self` is "declared" by the compiler, and does not have to be specified in the Structure Description.

[28] Alternatively, the result of a function may be specified by the RESULT declaration within the function itself.

5.4. Operator Overloading

GLISP provides operator overloading for user-defined objects using the Message facility. If an arithmetic operator is defined as the *selector* of a message for a user datatype, an arithmetic subexpression using that operator will be compiled as if it were a message call with two arguments. For example, the type VECTOR might have the declaration and function definitions below:

```
(GLISPOBJECTS
   (VECTOR  (CONS (X INTEGER) (Y INTEGER))
      MSC  ((+  VECTORPLUS OPEN T)
            (←+ VECTORINCR OPEN T)) )     )

(DEFINEQ

   (VECTORPLUS (GLAMBDA (U,V:VECTOR)
       (A VECTOR WITH X = U:X + V:X , Y = U:Y + V:Y) ))

   (VECTORINCR (GLAMBDA (U,V:VECTOR)
       (U:X ←+ V:X)
       (U:Y ←+ V:Y) ))     )
```

With these definitions, an expression involving the operators + or ←+ will be compiled by open compilation of the respective functions.

The compound operators (←+ +← ←- -←) are conventionally thought of as "destructive replacement" operators; thus, the expression (U ← U + V) will create a new VECTOR structure and assign the new structure to U, while the expression (U ←+ V) will smash the existing structure U, given the definitions above. The convention of letting the compound operators specify "destructive replacement" allows the user to specify both the destructive and non-destructive cases. However, if the compound operators are not overloaded but the arithmetic operators + and - are overloaded, the compound operators are compiled using the definitions of + for ←+ and +←, and - for ←- and -←. Thus, if only the + operator were overloaded for VECTOR, the expression (U ←+ V) would be compiled as if it were (U ← U + V).

5.5. Runtime Interpretation of Messages

In some cases, the type of the object which will receive a given message is not known at compile time; in such cases, the message must be executed interpretively, at runtime. Interpretive execution is provided for all types of GLISP messages.

An interpretive message call (i.e., a call to the function SEND) is generated by the GLISP compiler in response to a message call in a GLISP program when the specified message selector cannot be found for the declared type of the object receiving the message, or when the MESSAGE flag is set for that selector.

Alternatively, a call to SEND may be entered interactively by the user or may be contained in a function which has not been compiled by GLISP.

Messages can be interpreted only for those objects which are represented as one of the OBJECT types, since it is necessary that the object contain a pointer to its CLASS. The <selector> of the message is looked up in the MSG declarations of the CLASS; if it is not found there, the SUPERS of the CLASS are examined (depth-first) until the selector is found. The <response> associated with the <selector> is then examined. If the <response> is a function name, that function is simply called with the specified arguments.[29] If the <response> is a GLISP expression, the expression is compiled as a LAMBDA form and cached for future use.

Interpretive execution is available for other property types (PROP, ADJ, and ISA) using the call:

```
(SENDPROP <object> <selector> <proptype>)
```

where <proptype> is PROP, ADJ, or ISA. <proptype> is not evaluated.

[29] The object to which the message is sent is always inserted as the first argument, followed by the other arguments specified in the message call.

6. Context Rules and Reference

The ability to use definite reference to features of objects which are in *Context* is the key to much of GLISP's power. At the same time, definite reference introduces the possibility of ambiguity, i.e., there could be more than one object in Context which has a feature with a specified name. In this chapter, guidelines are presented for use of definite reference to allow the user to avoid ambiguity.

6.1. Organization of Context

The Context maintained by the compiler is organized in levels, each of which may have multiple entries; the sequence of levels is a stack. Searching of the Context proceeds from the top (nearest) level of the stack to the bottom (farthest) level. The bottom level of the stack is composed of the LAMBDA variables of the function being compiled. New levels are added to the Context in the following cases:

1. When a PROG is compiled. The PROG variables are added to the new level.

2. When a **For** loop is compiled. The "loop index" variable (which may be either a user variable or a compiler variable) is added to the new level, so that it is in context during the loop.

3. When a **While** loop is compiled.

4. When a new clause of an **If** statement is compiled.

When a Message, Property, or Adjective is compiled, that compilation takes place in a *new* context consisting only of the `self` argument and other message arguments.

6.2. Rules for Using Definite Reference

The possibility of referential ambiguity is easily controlled in practice. First, it should be noted that the traditional methods of unique naming and complete path specification ("PASCAL style") are available, and should be used whenever there is any possibility of ambiguity. Second, there are several cases which are guaranteed to be unambiguous:

1. In compiling GLISP code which implements a Message, Property, or Adjective, only the `self` argument is in context initially; definite reference to any substructure or property of the object is therefore unambiguous.[30]

2. Within a **For** loop, the loop variable is the closest thing in context.

3. In many cases, a function will only have a single structured argument; in such cases, definite

---

[30] Unless there are duplicated names in the object definition. However, if the same name is used as both a Property and an Adjective, for example, it is not considered a duplicate since Properties and Adjectives are specified by different source language constructs.

reference is unambiguous.

If "PASCAL" syntax (or the equivalent English-like form) is used for references other than the above cases, no ambiguities will occur.

6.3. Type Inference

In order to interpret definite references to features of objects, the compiler must know the *types* of the objects. However, explicit type specification can be burdensome, and makes it difficult to change types without rewriting existing type declarations. The GLISP compiler performs type inference in many cases, relieving the programmer of the burden of specifying types explicitly. The following rules enable the programmer to know when types will be inferred by the compiler.

1. Whenever a variable is set to a value whose type is known, the type of the variable is inferred to be the type of the value to which it was set.

2. If a variable whose initial type was NIL (e.g., an untyped PROG variable) appears on the left-hand side of the `<-+` operator, its type is inferred to be (LISTOF `<type>`), where `<type>` is the type of the right-hand side of the `<-+` expression.

3. Whenever a substructure of a structured object is retrieved, the type of the substructure is retrieved also.

4. Types of infix expressions are inferred.

5. Types of Properties, Adjectives, and Messages are inferred if:

   a. The `<response>` is GLISP code whose type can be inferred.

   b. The `<response>` has a RESULT declaration associated with it.

   c. The `<response>` is a function whose definition includes a RESULT declaration, or whose property list contains a GLRESULTTYPE declaration.

6. The type of the "loop variable" in a **For** loop is inferred and is added to a new level of Context by the compiler.

7. If an **If** statement tests the type of a variable using a `self` adjective, the variable is inferred to be of that type if the test is satisfied. Similar type inference is performed if the test of the type of the variable is the condition of a **While** statement.

8. When possible, GLISP infers the type of the function it is compiling and adds the type of the result to the property list of the function name under the indicator GLRESULTTYPE.

9. The types returned by many standard Lisp functions are known by the compiler.

7. GLISP and Knowledge Representation Languages

GLISP provides a convenient *Access Language* which allows uniform specification of access to objects, without regard to the way in which the objects are actually stored; in addition, GLISP provides a basic *Representation Language*, in which the structures and properties of objects can be declared. The field of Artificial Intelligence has spawned a number of powerful Representation Languages, which provide power in describing large numbers of object classes by allowing hierarchies of *Class* descriptions, in which instances of Classes can inherit properties and procedures from parent Classes. The *Access Languages* provided for these Representation Languages, however, have typically been rudimentary, often being no more than variations of LISP's GETPROP and PUTPROP. In addition, by performing inheritance of procedures and data values at runtime, these Representation Languages have often been computationally costly.

Facilities are provided for interfacing GLISP with representation languages of the user's choice. When this is done, GLISP provides a convenient and uniform language for accessing both objects in the Representation Language and LISP objects. In addition, GLISP can greatly improve the efficiency of programs which access the representations by performing lookup of procedures and data in the Class hierarchy *at compile time.* Finally, a LISP structure can be specified *as the way of implementing* instances of a Class in the Representation Language, so that while the objects in such a class appear the same as other objects in the Representation Language and are accessed in the same way, they are actually implemented as LISP objects which are efficient in both time and storage.

A clean[31] interface between GLISP and a Representation Language is provided. With such an interface, each *Class* in the Representation Language is acceptable as a GLISP *type.* When the program which is being compiled specifies an access to an object which is known to be a member of some Class, the interface module for the Representation Language is called to generate code to perform the access. The interface module can perform inheritance within the Class hierarchy, and can call GLISP compiler functions to compile code for subexpressions. Properties, Adjectives, and Messages in GLISP format can be added to Class definitions, and can be inherited by subclasses at compile time. In an Object-Centered representation language or other representation language which relies heavily on procedural inheritance, substantial improvements in execution speed can be achieved by performing the inheritance lookup at compile time and compiling direct procedure calls to inherited procedures when the procedures are static and the type of the object which inherits the procedure is known at compile time.

---

[31] Cleanliness is in the eye of the beholder and, being next to Godliness, difficult to attain. However, it's *relatively* clean.

Specifications for an interface module for GLISP are contained in a separate document[32]. To date, GLISP has been interfaced to our own GIRL representation language, and to LOOPS.[33]

---

[32] to be written.

[33] LOOPS, a LISP Object Oriented Programming System, is being developed at Xerox Palo Alto Research Center by Dan Bobrow and yMark Stefik.

8. Obtaining and Using GLISP

GLISP and its documentation are available free of charge over the ARPANET. The host computers involved will accept the login "ANONYMOUS GUEST" for transferring files with FTP.

8.1. Documentation

This user's manual, in line-printer format, is contained in [UTEXAS-20]<CS.NOVAK>GLUSER.LPT . The SCRIBE source file is [SU-SCORE]<CSD.NOVAK>GLUSER.MSS . Printed copies of this manual can be ordered from Publications Coordinator, Computer Science Department, Stanford University, Stanford, CA 94305, as technical report STAN-CS-82-895 ($3.15 prepaid); the printed version may not be as up-to-date as the on-line version.

8.2. Compiler Files

There are two files, GLISP (the compiler itself) and GLTEST (a file of examples). The files for the different Lisp dialects are:

Interlisp:	[SU-SCORE]<CSD.NOVAK>GLISP.LSP and GLTEST.LSP
Maclisp:	[SU-SCORE]<CSD.NOVAK>GLISP.MAC and GLTEST.MAC
UCI Lisp:	[UTEXAS-20]<CS.NOVAK>GLISP.UCI and GLTEST.UCI
ELISP:	the UCI version plus [UTEXAS-20]<CS.NOVAK>ELISP.FIX
Franz Lisp:	[SUMEX-AIM]<NOVAK>GLISP.FRANZ and GLTEST.FRANZ
PSL:	[SU-SCORE]<CSD.NOVAK>GLISP.PSL and GLTEST.PSL

8.3. Getting Started

Useful functions for invoking GLISP are:

(GLCC 'FN)	Compile FN.
(GLCP 'FN)	Compile FN and prettyprint result.
(GLP 'FN)	Prettyprint GLISP-compiled version of FN.
(GLED 'NAME)	Edit the property list of NAME.
(GLEDF 'FN)	Edit the original (GLISP) definition of FN. (The original definition is saved under the property "GLORIGINALEXPR" when the function is compiled, and the compiled version replaces the function definition.)
(GLEDS 'STR)	Edit the structure declarations of STR.

The editing functions call the "BBN/Interlisp" structure editor.

To try out GLISP, load the GLTEST file and use GLCP to compile the functions CURRENTDATE, GIVE-RAISE, TESTFN1, TESTFN2, DRAWRECT, TP, GROWCIRCLE, and SQUASH. To run compiled functions on test data, do:

```
(GIVE-RAISE 'COMPANY1)
(TP '(((A (B (C D (E (G H (I J (K)))))))))
(GROWCIRCLE MYCIRCLE)
```

8.4. Reserved Words and Characters

GLISP contains ordinary lisp as a sublanguage. However, in order to avoid having code which was intended as "ordinary lisp" interpreted as GLISP code, it is necessary to follow certain conventions when writing "ordinary lisp" code.

8.4.1. Reserved Characters

The colon and the characters which represent the arithmetic operators should not be used within atom names, since GLISP splits apart "atoms" which contain operators. The set of characters to be avoided within atom names is:

```
+ * / ↑ ← } = < > : ' ,
```

The character "minus" (-) is permitted within atom names unless the flag GLSEPMINUS is set.

Some GLISP constructs permit (but do not require) use of the character "comma" (,); since the comma is used as a "backquote" character in some Lisp dialects, the user may wish to avoid its use. When used in Lisp dialects which use comma as a backquote character, all commas must be "escaped" or "slashified"; this makes porting of GLISP code containing commas more difficult.

8.4.2. Reserved Function Names

Most GLISP function, variable, and property names begin with "GL" to avoid conflict with user names. Those "function" names which are used in GLISP constructs or in interpretive functions should be avoided. This set includes the following names:

```
A          AN         CASE       FOR        IF
REPEAT     SEND       SENDPROP   THE        WHILE
```

8.4.3. Other Reserved Names

Words which are used within GLISP constructs should be avoided as variable names. This set of names includes:

```
A          AN         DO         ELSE       ELSEIF
IS         OF         THE        THEN       UNTIL
```

8.5. Lisp Dialect Idiosyncrasies

GLISP code passes through the Lisp reader before it is seen by GLISP. For this reason, operators in expressions may need to be set off from operands by blanks; the operator "-" should always be surrounded by blanks, and the operator "+" should be separated from numbers by blanks.

8.5.1. Interlisp

GLISP compilation happens automatically, and usually does not need to be invoked explicitly. GLISP declarations are integrated with the file package.

8.5.2. UCI Lisp

The following command is needed before loading to make room for GLISP:

```
(REALLOC 3000 1000 1000 1000 35000)
```

The compiler file modifies the syntax of the character ~ to be "alphabetic" so it can be used as a GLISP operator. The character "/" must be "slashified" to "//".

8.5.3. ELISP

For ELISP, the UCI Lisp version of the compiler is used, together with a small compatibility file. The above comments about UCI lisp do not apply to ELISP. The characters "/" and "," must be "slashified" to "//" and "/,".

8.5.4. Maclisp

The characters "/" and "," must be "slashified" to "//" and "/,".

8.5.5. Franz Lisp

Automatic compilation is implemented for Franz Lisp. The character "," and the operators "+←" and "-←" must be "slashified" to "\,", "+\←", and "-\←", respectively. Before loading GLISP, edit something

to cause the editor files to be loaded[34]. The Franz Lisp version of GLISP has been tested on Opus 38 Franz Lisp; users with earlier versions of Franz might encounter difficulties.

8.6. Bug Reports and Mailing List

To get on the GLISP mailing list or to report bugs, send mail to CSD.NOVAK@SU-SCORE.

[34]Some versions of the "CMU editor" contain function definitions which may conflict with those of GLISP; if the editor is loaded first, the GLISP versions override.

9. GLISP Hacks

This chapter discusses some ways of doing things in GLISP which might not be entirely obvious at first glance.

9.1. Overloading Basic Types

GLISP provides the ability to define properties of structures described in the Structure Description language; since the elementary LISP types are structures in this language, objects whose storage representation is an elementary type can be "overloaded" by specifying properties and operators for them. The following examples illustrate how this can be done.

```
(GLDEFSTRQ

(ArithmeticOperator  (self ATOM)

   PROP ((Precedence OperatorPrecedenceFn  RESULT INTEGER)
         (PrintForm  ((GETPROP self 'PRINTFORM) or self)) )

   MSG  ((PRIN1      ((PRIN1 the PrintForm)))) )

(IntegerMod7         (self INTEGER)

   PROP ((Modulus    (7))
         (Inverse    ((If self is ZERO then 0
                          else (Modulus - self))) ))

   ADJ  ((Even       ((ZEROP (LOGAND self 1))))
         (Odd        (NOT Even)))

   ISA  ((Prime      PrimeTestFn))

   MSG  ((+          IMod7Plus  OPEN T  RESULT IntegerMod7)
         (←          IMod7Store OPEN T  RESULT IntegerMod7)) )

)
(DEFINEQ

(IMod7Store  (GLAMBDA (LHS:IntegerMod7 RHS:INTEGER)
        (LHS:self ← (IREMAINDER RHS Modulus)) ))

(IMod7Plus   (GLAMBDA (X,Y:IntegerMod7)
        (IREMAINDER (X:self + Y:self) X:Modulus) ))
)
```

A few subtleties of the function IMod7Store are worth noting. First, the left-hand-side expression used in storing the result is LHS:self rather than simply LHS. LHS and LHS:self of course refer to the same actual structure; however, the *type* of LHS is IntegerMod7, while the type of LHS:self is INTEGER. If LHS were

used on the left-hand side, since the ← operator is overloaded for IntegerMod7, the function IMod7Store would be invoked again to perform its own function; since the function is compiled OPEN, this would be an infinite loop. A second subtlety is that the assignment to LHS:self must use the self-assignment operator, ←← , since it is desired to perform assignment as seen "outside" the function IMod7Store, i.e., in the environment in which the original assignment operation was specified.

9.2. Disjunctive Types

LISP programming often involves objects which may in fact be of different types, but which are for some purposes treated alike. For example, LISP data structures are typically constructed of CONS cells whose fields may point to other CONS cells or to ATOMs. The GLISP Structure Description language does not permit the user to specify that a certain field of a structure is a CONS cell *or* an ATOM. However, it is possible to create a GLISP datatype which encompasses both. Typically, this is done by declaring the structure of the object to be the complex structure, and testing for the simpler structure explicitly. This is illustrated for the case of the LISP tree below.

```
(LISPTREE  (CONS (CAR LISPTREE) (CDR LISPTREE))

   ADJ     ((EMPTY      (~self)))

   PROP    ((LEFTSON    ((If self is ATOMIC then NIL else CAR)))
            (RIGHTSON   ((If self is ATOMIC then NIL else CDR)))))
```

9.3. Generators

Often, one would like to define such properties of an object as the way of enumerating its parts in some order. Such things cannot be specified directly as properties of the object because they depend on the previous state of the enumeration. However, it is possible to define an object, associated with the original datatype, which contains the state of the enumeration and responds to Messages. This is illustrated below by an object which searches a tree in Preorder.

```
(PreorderSearchRecord  (CONS (Node LISPTREE)
                             (PreviousNodes (LISTOF LISPTREE)))

   MSG  ((NEXT  ((PROG (TMP)
                       (If TMP←Node:LEFTSON
                           then (If Node:RIGHTSON
                                    then PreviousNodes←←Node)
                                Node←TMP
                           else TMP←←PreviousNodes
                                Node←TMP:RIGHTSON) ))))


(TP (GLAMBDA ((A LISPTREE))
      (PROG (PSR)
         (PSR ← (A PreorderSearchRecord
                   with Node = (the LISPTREE)))
         (While Node (If Node is ATOMIC (PRINT Node))
                     (← PSR NEXT)) )))
```

The object class PreorderSearchRecord serves two purposes: it holds the state of the enumeration, and it responds to messages to step through the enumeration. With these definitions, it is easy to write a program involving enumeration of a LISPTREE, as illustrated by the example function TP above. By being open-compiled, messages to an object can be as efficient as in-line hand coding; yet, the code for the messages only has to be written once, and can easily be changed without changing the programs which use the messages.

10. Program Examples

In this chapter, examples of GLISP object declarations and programs are presented. Each example is discussed as a section of this chapter; the code for the examples and the code produced by the compiler are shown for each example at the end of the chapter.

10.1. GLTST1 File

The GLTST1 file illustrates the use of several types of LISP structures, and the use of fairly complex Property definitions for objects. SENIORITY of an EMPLOYEE, for example, is defined in terms of the YEAR of DATE-HIRED, which is a substructure of EMPLOYEE, and the YEAR of the function (CURRENTDATE).[35]

10.2. GLTST2 File

The GLTST2 file illustrates the use of Messages for ordinary LISP objects. By defining the arithmetic operators as Message selectors for the object VECTOR, use of vectors in arithmetic expressions is enabled; OPEN compilation is specified for these messages.

The definition of GRAPHICSOBJECT uses VECTORs as components. While the actual structure of a GRAPHICSOBJECT is simple, numerous properties are defined for user convenience. The definition of CENTER is easily stated as a VECTOR expression.

The Messages of GRAPHICSOBJECT illustrate how different responses to a message for different types of objects can be achieved, even though for GLISP compilation of messages to LISP objects the code for a message must be resolved at compile time.[36] The DRAW and ERASE messages get the function to be used from the property list of the SHAPE name of the GRAPHICSOBJECT and APPLY it to draw the desired object.

MOVINGGRAPHICSOBJECT contains a GRAPHICSOBJECT as a TRANSPARENT component, so that it inherits the properties of a GRAPHICSOBJECT; a MOVINGGRAPHICSOBJECT is a GRAPHICSOBJECT which has a VELOCITY, and will move itself by the amount of its velocity upon the

---

[35] The *type* of (CURRENTDATE) must be known to the compiler, either by compiling it first, or by including a RESULT declaration in the function definition of CURRENTDATE, or by specifying the GLRESULTTYPE property for the function name.

[36] For objects in a Representation Language, messages may be compiled directly as LISP code or as messages to be interpreted at runtime, depending on how much is known about the object to which the message is sent and the compilation declarations in effect.

message command STEP.[37] The compilation of the message (← MGO STEP) in the function TESTFN1 is of particular interest. This message is expanded into the sending of the message (← self MOVE VELOCITY) to the MOVINGGRAPHICSOBJECT. The MOVINGGRAPHICSOBJECT cannot respond to such a message; however, since it contains a GRAPHICSOBJECT as a TRANSPARENT component, its GRAPHICSOBJECT responds to the message.[38] A GRAPHICSOBJECT responds to a MOVE message by erasing itself, increasing its START point by the (vector) distance to be moved, and then redrawing itself. All of the messages are specified as being compiled open, so that the short original message actually generates a large amount of code.

A rectangle is drawn by the function DRAWRECT. Note how the use of the properties defined for a GRAPHICSOBJECT allows an easy interface to the system functions MOVETO and DRAWTO in terms of the properties LEFT, RIGHT, TOP, and BOTTOM.

---

[37] This example is adapted from the MovingPoint example written by Dan Bobrow for LOOPS.

[38] TRANSPARENT substructures thus permit procedural inheritance by LISP objects.